Chapter 1

Shearsort

Shearsort is an algorithm for sorting P keys on a $\sqrt{P} \times \sqrt{P}$ mesh. We first analyze the deterministic case, then the average case. Finally, we consider a slight modification of shearsort which turns out to be efficient for the case of $N \gg P$ keys on P processors.

1.1 Basic Algorithm

Consider a mesh of $\sqrt{P} \times \sqrt{P}$ processors each holding a single key. At the end the elements are arranged in a snake like order,i.e. the smallest to largest in processors $11, 12, \ldots, 1P, 2P, 2P - 1, \ldots, 21, 31, \ldots, PP$. The algorithm is:

Phase 1: Do k times
sort odd numbered lows left to right and even numbered rows right to left.
sort columns top to bottom.

Phase 2: Sort rows alternating as above.

Each row and column sorting step takes $O(\sqrt{P})$ time. We will show below that it is enough to choose $k = \log \sqrt{P}$, so the total time is $O(\sqrt{P} \log P)$. The correctness argument is as follows.

First, note that the algorithm is an oblivious comparison exchange algorithm. Hence the zero-one lemma applies. So we assume that the input consists of zeros and ones only. Call a row *clean* if it contains only zeros or only ones. Otherwise a row is *dirty*. Initially all rows can be dirty.

We first argue that after one iteration of the first phase, the number of dirty rows can be at most $n/2$. After the row sorting operation, the keys in the first row will be organized as per the pattern $0^i 1^j$ and those in the second as per $1^m 0^n$ because of the alternate sorting direction. Consider now what happens after the first step of the column sorting operation. In this step, the elements in the first row are with the corresponding elements of the second row, and the smaller retained above and the larger below. If $i = m$ then the upper row will become a clean row of zeros and the lower a clean row of ones. If $i < m$ then the lower row will become a clean row of ones, while if $i > m$ the upper row a clean row of zeros. Thus at this point at least one clean row will be produced. But this will happen in every pair of rows. Further, the subsequent steps of the column sorting operation will not destroy any clean row; it will only cause all zero clean rows to move to the top and the one clean rows to move to the bottom. Hence after one iteration, there will be at most $n/$ dirty rows, which will be placed below the zero clean rows, and above the one clean rows.

Now the same argument will apply in each iteration and hence after $k = \log \sqrt{P}$ only one dirty row will remain. The second phase will sort this dirty row, and it is easily verified that this will result in keys to be sorted in the snake like order described earlier.

1.2 Average Case

Theorem 1 *Suppose input to ShearSort is a random permutation of* $(1 \dots P)$*. Let* k *be a random variable denoting the number of iterations needed by Shearsort. Then* $\mathbf{E}[k] = \Omega(\log P)$*.*

Before we prove this theorem, let us consider the deterministic case. Consider the following input to ShearSort:

$$\begin{bmatrix} 0 & 1 & \dots & 1 \\ 0 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 1 \end{bmatrix}$$

i.e. $\sqrt{P}$ 0s and rest 1s initially arranged as shown above. After each iteration of the algorithm, the number of *dirty* rows is at most halved. So $k = \Omega(\log P)$.

In general, if the $\sqrt{P}$ 0s are placed in some q rows at the beginning, $\log q$ iterations will be needed because the number of dirty rows will not reduce faster than halving at each step. We build up on this idea.

Proof of Theorem 1: Let s be a permutation of $1..P$. Let $f(s)$ denote the sequence obtained by mapping f onto s, where f is defined as:

$$f(x) = \begin{cases} 1 & \text{if } x > \sqrt{P} \\ 0 & \text{otherwise} \end{cases}$$

Clearly, if Shearsort correctly sorts s, then it must also correctly sort $f(s)$. We will show that sorting $f(s)$ must take a large number of iterations on the average.

The main point to show is that when s is chosen at random, the expected number of rows containing at least one 0 after applying f is high. But these dirty rows will only halve in each iteration, and hence a large number of iterations will be needed.

When the input is s, the probability that the smallest $\sqrt{P}$ elements in s all lie outside any given row is atmost

$$\frac{P-\sqrt{P}}{P}\frac{P-\sqrt{P}-1}{P-1}\cdots\frac{P-2\sqrt{P}+1}{P-\sqrt{P}+1} \leq \left(\frac{P-\sqrt{P}}{P}\right)^{\sqrt{P}} = \left(1-\frac{1}{\sqrt{P}}\right)^{\sqrt{P}} \leq \frac{1}{e}$$

Hence after applying f, the probability that any give row has atleast one 0 is at least $1-\frac{1}{e} = 0.63$. Let X_i be an indicator random variable taking value 0 if row i contains no 0, and value 1 otherwise. Clearly

$$\mathbf{E}[X_i] = Pr[X_i = 1] \geq 1 - \frac{1}{e}$$

Let X =random variable denoting the number of rows having atleast one 0 after f is applied. Since $X = \sum_{i=1}^{\sqrt{P}} X_i$ we have

$$\mathbf{E}[X] \geq \sqrt{P}(1 - \frac{1}{e}) = \Omega(\sqrt{P})$$

The expected number of iterations is thus $\log \mathbf{E}[\log X] = \Omega(\log P)$. Thus, even on the average the number of iterations is $\log P$ to within a constant factor.

1.2.1 P processors, N keys

Now consider a mesh of $\sqrt{P} \times \sqrt{P}$ processors with $L = \frac{N}{P}$ keys per processor. Consider a modified algorithm in which column sorting happens before row sorting:

For $i = 1$ to k
Sort columns top to bottom
Sort odd numbered rows left to right and even numbered right to left.

We will see later that each row or column sorting step will take time $L\sqrt{P}$. Thus total time is $O(L \log L + kL\sqrt{P})$. The important question is: *for random input, does small k suffice?*

Theorem 2 *Suppose each processor independently generates L numbers in the range* $1..R$. *Then with high probability the above algorithm correctly sorts using only 2 iterations, provided* $L \geq 4P$ *and* $R = P$.

Proof: We will show that with high probability, after the first column sorting step, there will be at most 2 dirty rows. Thus a row sorting step following a column sorting step will reduce it to on. The final row sorting step will finish the sorting.

Let s denote the input to the algorithm. Define $f_i(x) = 0$ if $x \leq i$ and $f_i(x) = 1$ otherwise. Let $f_i(s)$ denote the sequence obtained. Clearly, the algorithm is correct on s if it is correct for all $f_i(s)$. We first estimate for any fixed i the probability that more than 2 rows are dirty after the first column sort. Define $\alpha_i = \frac{i}{R}$ i.e. the probability that a number less than i is chosen in any position.

Let X_j = number of 0s in jth column. Then $E[X_j]$ = number of keys in jth column times $\alpha_i = \sqrt{P}L\alpha_i$. If $X_j = \sqrt{P}L\alpha_i$ for all j, then $f_i(s)$ would be sorted after just one iteration! Of course, this is unlikely. But we will show that X_j will be within $L/2$ of its expectation. This will show that the first column sort will leave at most 2 dirty rows.

Notice that X_j is a sum of independent zero-one random variables, and thus Chernoff bounds apply. For estimating the probability that the number of zeros is smaller than the expectation, we use the Chernoff bounds $Pr[X \leq (1 - \epsilon)E[X]] \leq e^{-\epsilon^2 E[X]/2}$ for $0 < \epsilon$. In our case $E[X_j] \leq L\sqrt{P}$ and $\epsilon = L/2E[X_j]$. Thus

$$Pr[X_j \leq E[X_j] - L/2] \leq e^{-L/8\sqrt{P}}$$

To show that X_j cannot be much larger than $E[X_j]$ we need two cases. Note first that for $0 < \epsilon < 2e - 1$ we have $Pr[X \geq (1 + \epsilon)E[X]] \leq e^{-\epsilon^2 E[X]/4}$. Thus for $\epsilon < 2e - 1$ we will have

$$Pr[X_j \geq E[X_j] + L/2] \leq e^{-L/16\sqrt{P}}$$

For $2e - 1 \leq \epsilon$ we have $Pr[X \geq (1 + \epsilon)E[X]] \leq 2^{-(1+\epsilon)\mu}$. Thus we have

$$Pr[X_j \geq E[X_j] + L/2] \leq 2^{-L/2 - E[X_j]} \ll e^{-L/16\sqrt{P}}$$

Thus assuming $L \geq 16P$ we have

$$Pr[|X_j - E[X_j]| \geq L/2] \leq 2e^{-\sqrt{P}}$$

Now, we need that no X_j is away from its expected value by more than $L/2$, for all $j = 1 \ldots \sqrt{P}$. And as mentioned earlier, also for all i in the range 1 through R. Thus the probability that there are more than 2 dirty rows for input s after the first column sort is at most

$$2e^{-\sqrt{P}} R\sqrt{P}$$

which is very small for $R = P$. ■

1.2.2 Time Analysis

We require the $\sqrt{P}$ processors in each row to sort $L\sqrt{P}$ numbers. This can be done as follows:

1. Each processor locally sorts the L numbers it has.
2. Next processors execute a modified odd-even transposition sort: instead of a comparison exchange step, processors exchange all the keys, merge them and then the lower numbered processor retains the smaller L, and the larger numbered processor the larger L.

That this will sort correctly is an exercise. In fact, the above idea works in general: given any comparison exchange algorithm with 1 key per processor, we can obtain an algorithm for the L keys per processor case by adding a local sorting step and using a merge-split step as above.

The time taken is the $O(L \log L + L\sqrt{P})$ for single row or column sort. For consecutive sorting operations, the local sorting does not need to be repeated, since the preceding sorting operations leave the elements in a processor sorted.

1.2.3 Lower bounds:

Using the bisection bound, the time must be at least $L\sqrt{P}$. Since the algorithm executes for only $O(1)$ iterations, the lower bound is matched.

Exercises

1. We are given a linear array of processors with one key per processor. Suppose that every key is at most a distance d from its final position. Show that the array can be sorted by using at most $O(d)$ steps. Estimate the constant in $O(d)$.

   (Will the zero-one lemma immediately apply here? Clarify/extend as needed. Then use it.)

2. The quadrant sort algorithm also takes 1 key per processor in a 2 dimensional mesh and sorts in a snake like final order. It works as follows:

   (a) Recursively sort the quadrants (in snake like order)

(b) Do 3 times:
Sort rows alternately
Sort columns vertically.

(c) Sort rows alternately.

Prove that this will correctly sort. Estimate the time taken.

3. Show that in light of exercise 1, the algorithm of exercise 2 can be simplified. This will not change the order of the algorithm, but will change the constant factors. Determine the old and new expressions for the running time including the constant factors for the $O(\sqrt{n})$ term. (The array is $\sqrt{n} \times \sqrt{n}$.)

4. In the average case analysis, are 2 iterations really needed? Characterize input instances for which 2 iterations will be needed, and examine if they are likely.